The Common Component Architecture for Scalable Scientific Software Engineering

Kosta Damevski

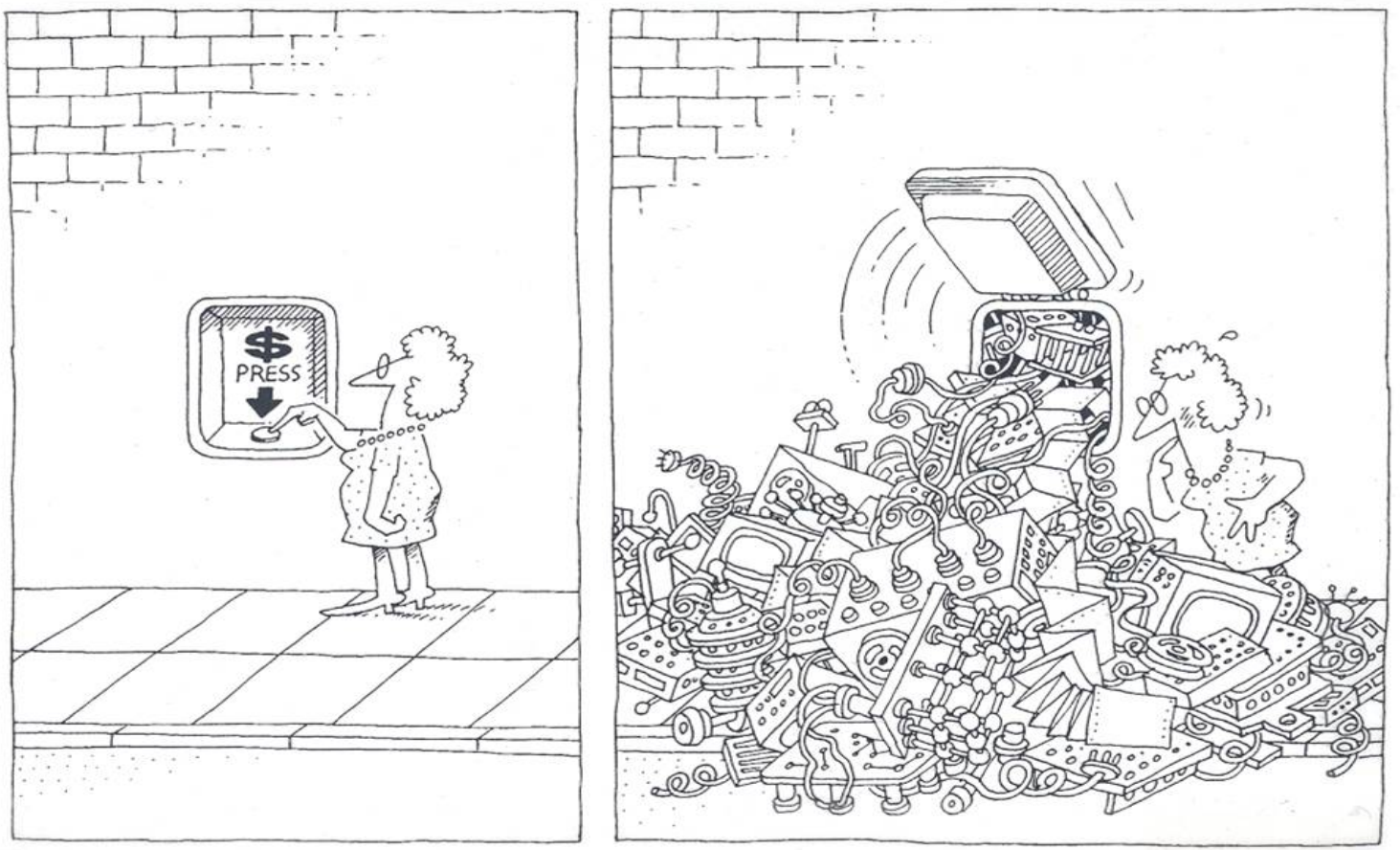
Department of Mathematics and Computer Science
Virginia State University

September 22, 2009

Outline

Outline

Why Components?



The task of the software development team is to engineer the illusion of simplicity [Booch]

Complexity in Scientific Software Development

- At least 41 different Fast Fourier Transform (FFT) libraries
  - see: http://www.fftw.org/benchfft/ffts.html
- Many (if not all) have different interfaces
  - Different procedure names and different input and output parameters

For instance...

```
SUBROUTINE FOUR1(DATA, NN, ISIGN)

Replaces DATA by its discrete Fourier transform
(if ISIGN is input as 1) or replaces DATA by NN
times its inverse discrete Fourier transform
(if ISIGN is input as -1)...
```

- By using components, a common interface would allow plug and play FFT libraries

Background

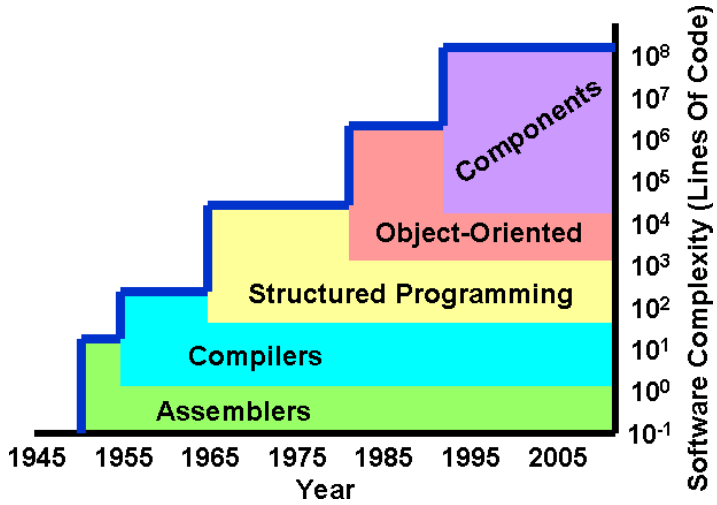
- High-performance computational science is now widely recognized as critical to scientific discovery.
- Researchers face numerous challenges to exploiting emerging leadership-class systems for multi-model simulations at extreme scale
  - Scalability in problem size/complexity, resources (CPUs, memory, I/O)
  - **Also software size/complexity/rate of change, human factors**

Scientific Software Development

- Software is a team effort
  - Geographically distributed
  - Multidisciplinary
- Software must be robust, but flexible
  - Much larger than any single contributor can deeply understand
  - Much longer lifetime than specific platforms, contributors
  - Allow evolution of algorithms, scientific focus

Components Help Manage Software Complexity

- Extend object-oriented ideas (abstraction, encapsulation, separation of concerns, modularity) with **composition**



  - Components **encapsulate functionality** which is exposed to the outside through well-defined **interfaces**
  - Applications formed by composing components based on their interfaces (type matching)

Benefits of Components for Scientific Computing

- Tool to organize large, complex software systems in terms of smaller modules of manageable scale
- Components map well onto individuals or small groups of developers
- Interfaces between components map well onto interfaces between development groups, geographic location, scientific specialty, programming language preferences, etc.
- *Goal: Enable the scientist to do more science and less software development*

The Common Component Architecture (CCA)

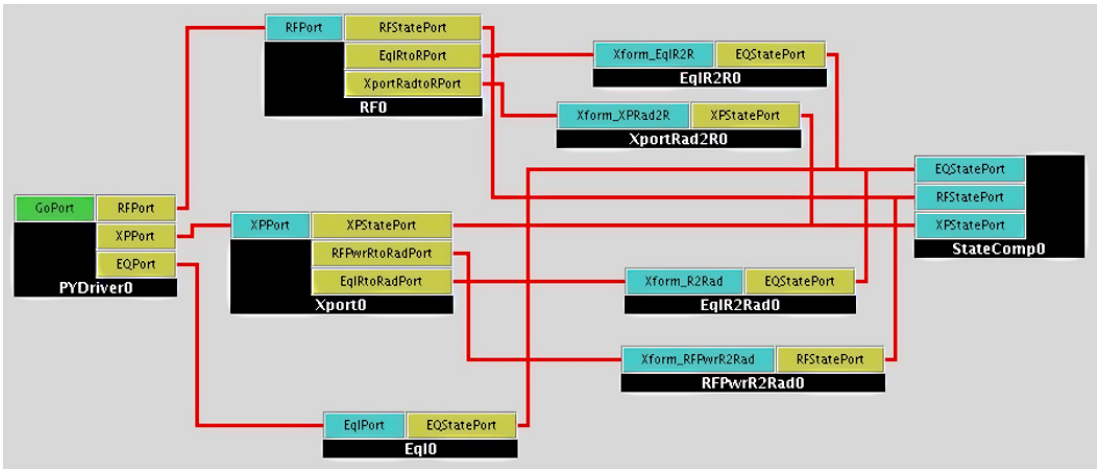
- A component architecture specially designed for high-performance scientific computing
  - “Commodity” approaches not sufficient
- Supports parallel and distributed computing
- Supports mixed language programming
  - Currently: C, C++, Fortran, Java, Python
- Support for platforms, data types, etc. important to HPC
- Support for legacy software

Outline

Basic CCA Concepts and Terms

Components

- Units of software development/functionality
- Interact only through well-defined interfaces
- Can be composed into applications based on their interfaces



Screenshot of an application in the Ccaffeine framework GUI

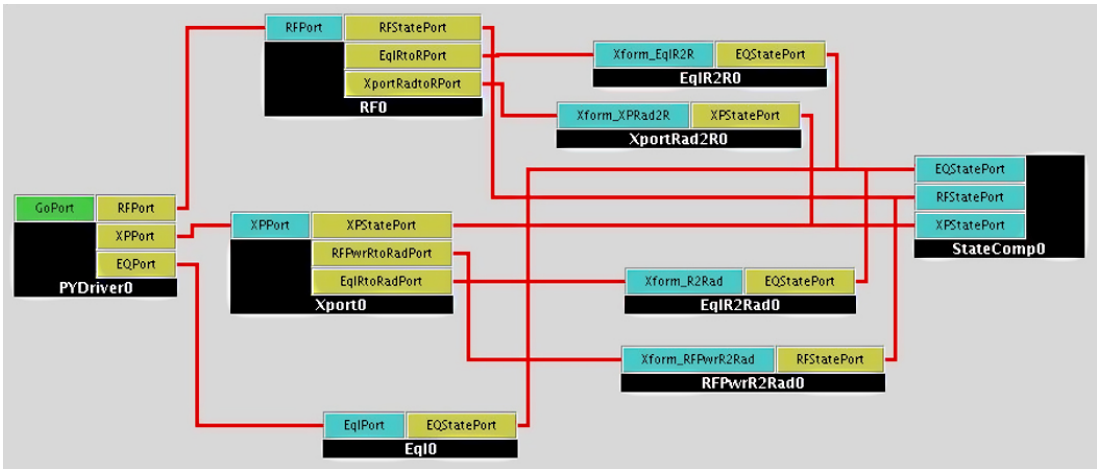
Basic CCA Concepts and Terms

Components

- Units of software development/functionality
- Interact only through well-defined interfaces
- Can be composed into applications based on their interfaces

Ports

- The interfaces through which components interact
- Follow a provides/uses pattern



Screenshot of an application in the Ccaffeine framework GUI

Basic CCA Concepts and Terms

Components

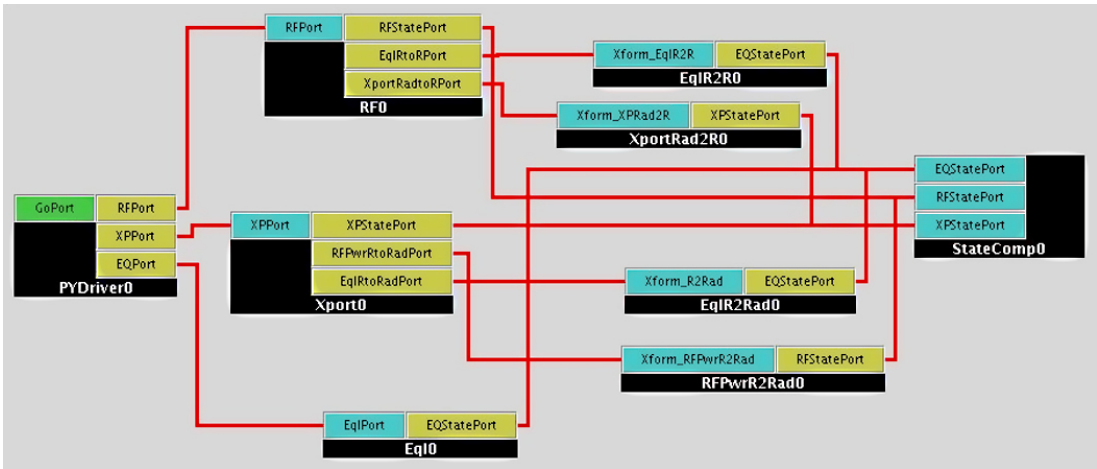
- Units of software development/functionality
- Interact only through well-defined interfaces
- Can be composed into applications based on their interfaces

Frameworks

- Hold components while applications are assembled and executed
- Control the connections of ports
- Provide standard services to components

Ports

- The interfaces through which components interact
- Follow a provides/uses pattern



Screenshot of an application in the Ccaffeine framework GUI

The CCA Specification is the 'A' in CCA

- 23 interfaces define rules and responsibilities for components and frameworks
- CCA places a small conceptual burden on users

Interfaces a typical user must understand

- gov.cca.Port
- gov.cca.Component
- gov.cca.Services
- gov.cca.CCAException

Minimum interfaces required to implement a CCA framework

- gov.cca.AbstractFramework
- gov.cca.Services
- gov.cca.ComponentID
- gov.cca.TypeMap
- (and a few exception interfaces)

CCA Forum Governs The CCA Specification

- Formed (in 1997) as a grass roots effort by individuals from:
  - Several DoE labs (Los Alamos, Sandia, Lawrence Livermore, Oak Ridge)
  - NASA
  - Indiana University
  - University of Utah
- Forum membership has expanded to a number of private and public institutions
  - Funded by both SCIDAC 1 and 2
- Quarterly meetings
  - Attendance in the last two out of last three meetings is required for voting

CCA Software Responsibilites

Implemented By CCA Users

- Interfaces (ports) for components
- Components
- Applications (assemblies of components)

Provided By CCA

- CCA Specification
  - Interfaces for standard services
  - Implementations of standard services (as components)
- Scientific Interface Definition Language (SIDL)
- Frameworks
  - Ccaffeine, Decaf, SCIJump (SCIRun2), XCAT
- Component Development Tools
  - Babel, Bocca, OnRamp

Component Lifecycle

1. Implement component
   - Express ports and component in a Scientific Interface Definition Language (SIDL)
   - Compile SIDL with Babel compiler
   - Provide implementation
     - ★ The component expresses the ports it *provides* and the ones it *uses*

Component Lifecycle

1. Implement component
   - Express ports and component in a Scientific Interface Definition Language (SIDL)
   - Compile SIDL with Babel compiler
   - Provide implementation
     - ★ The component expresses the ports it *provides* and the ones it *uses*
2. Instantiate component in a framework
   - Upon instantiation, each component has its ports discovered by the framework

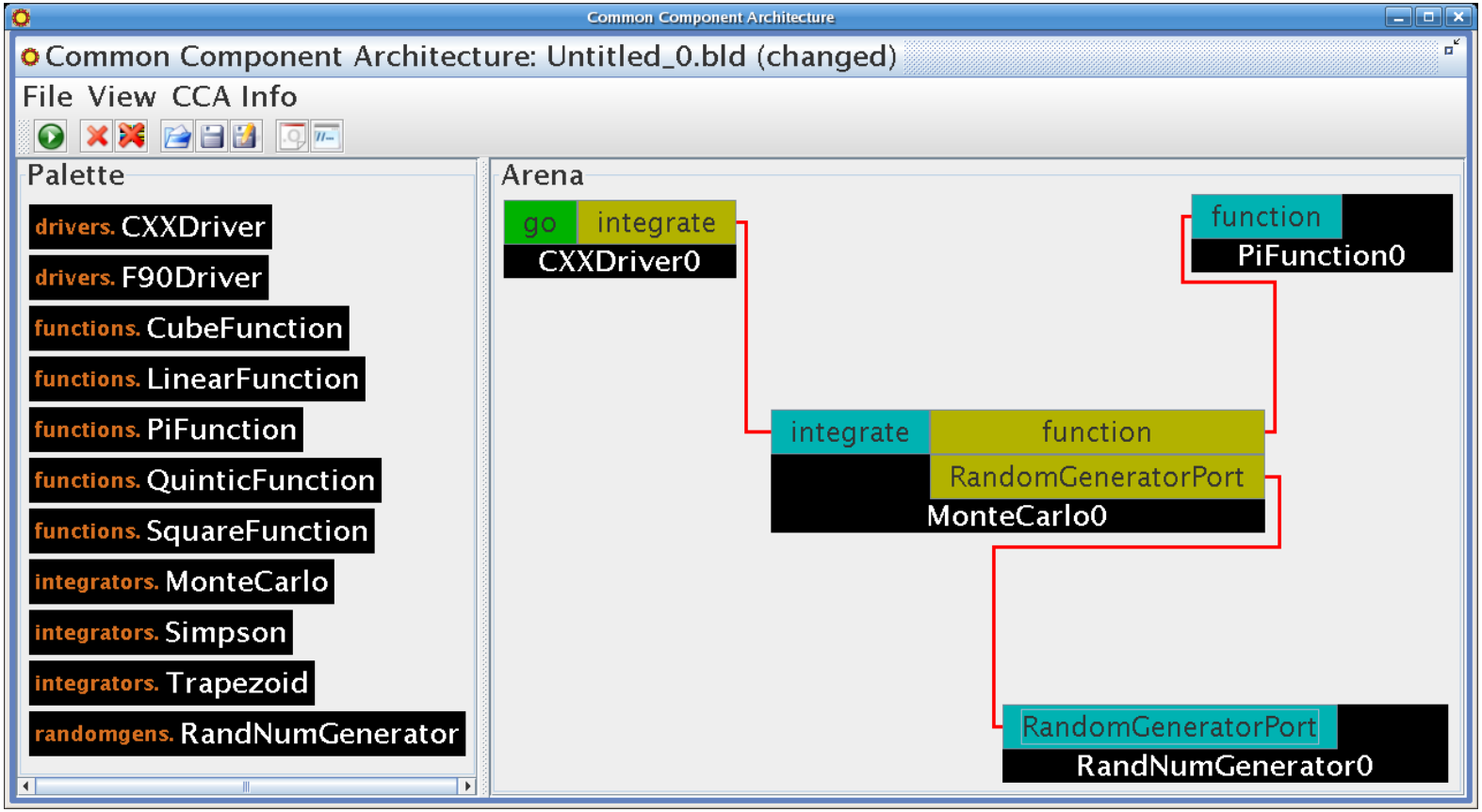
Component Lifecycle

1. Implement component
   - Express ports and component in a Scientific Interface Definition Language (SIDL)
   - Compile SIDL with Babel compiler
   - Provide implementation
     - The component expresses the ports it *provides* and the ones it *uses*
2. Instantiate component in a framework
   - Upon instantiation, each component has its ports discovered by the framework
3. Connect components (composed) to form an application
   - Often done through the framework's GUI

Component Lifecycle

1. Implement component
   - Express ports and component in a Scientific Interface Definition Language (SIDL)
   - Compile SIDL with Babel compiler
   - Provide implementation
     - ★ The component expresses the ports it *provides* and the ones it *uses*
2. Instantiate component in a framework
   - Upon instantiation, each component has its ports discovered by the framework
3. Connect components (composed) to form an application
   - Often done through the framework's GUI
4. Instruct the application to begin
   - Begin execution at component that provides a special port called a *GoPort*
   - After the instruction to begin, the rest of the application's control flow is handled by the components themselves
   - Components communicate via method invocation

A Simple Integration Example



Four components (Driver, Integrator, Function and RandomNumberGenerator) are composed (in the Ccaffeine GUI)

Express Interfaces in SIDL

SIDL for Integration port

```
package integrator version 0.1 {
  interface IntegratorPort extends gov.cca.Port
  {
    double integrate(in double lowBound,
                     in double upBound,
                     in int count);
  }
}
```

- Scientific Interface Definiton Language (SIDL) is used to express component interfaces
  - Loosely based on the CORBA IDL
- SIDL is compiled by the Babel (LLNL) compiler

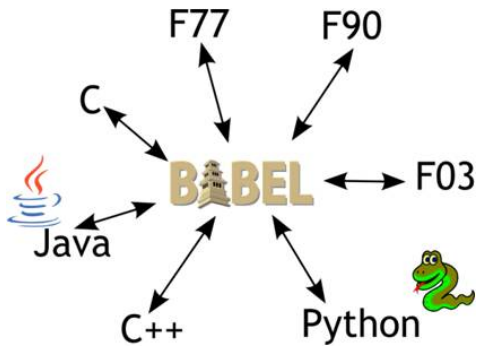
A General Idea of Component Implementation

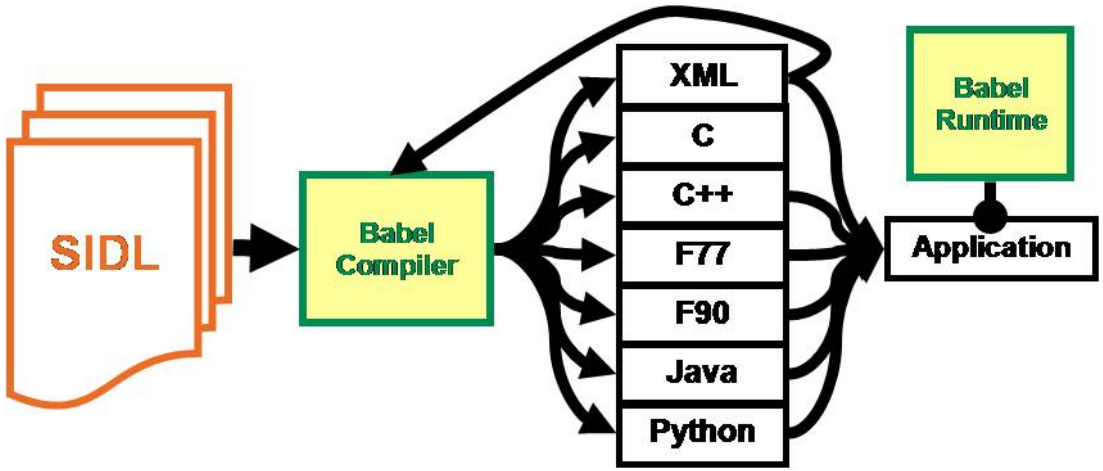
Commonly Used Methods

- *addProvidesPort*
  - Component informs the framework about a port it provides
- *registerUsesPort*
  - Component informs the framework about a port it uses
- *connect*
  - Connects a provides to a uses port (often done by the framework through a UI)
    - ★ Port compatibility is checked
- *getPort*
  - Once two ports are connected, a component with a uses port can use this method to get a handle to the provides port
    - ★ Handle is used to make procedure/method calls

Babel is a Foundational Part of the CCA

- Babel provides...
  - Fast inter-language communication for C, C++, Fortran, Python, and Java
  - Consistent OO type system
  - Support for distributed communication via Remote Method Invocation (RMI)
  - Interface contract specifications





Frameworks Manage Component Instances

- Able to create and compose components into an application
- Provide components a discovery mechanism, and a few useful services
- Frameworks interfere minimally in the execution of a component
  - Scale up easily to thousands of component instances
- As long as a framework is based on the CCA specification, it can always add more functionality
  - Ccaffeine (Sandia) provides a GUI and support for SPMD parallel components, and is the only framework that is at “production” level
  - SCIJump (UofU, VSU) provides similar features, but parallel component concept is more MPMD
  - XCAT (IU, BU), Decaf (LLNL)

Support for Distributed and Parallel Computing in CCA

Distributed

- Babel provides the basis for distributed computing
  - Several wire protocols available
- Some frameworks support distributed components (SCIRun2, XCAT)

Parallel

- Component model allows for MPI (or PVM, etc.) based components
- CCaffeine framework manages parallel components in a SCMD fashion
- Parallel coupling support is available through components (InterComm, GlobalArrays)

CCA Toolkit

Goal: *Create a repository of high-quality, high-performance components that enable scientific developers to perform rapid prototyping, and access widely used libraries in component form*

- Access to diverse linear solvers developed at different institutions: hypre, MUMPS, PETSc, SPARSKIT, SuperLU, Trilinos, ...
- InterComm (UMD) component provides parallel model coupling capabilies
- Performance monitoring: Tuning and Analysis Utility - TAU (UO), PerfExplorer (UO)
- Numerical optimization: Toolkit for Advanced Optimization - TAO (ANL)
- Global Array (PNNL) provides a global-view parallel programming model
- Parallel Cartesian Grid (SNL): Simple 2d Cartesian mesh
- (a number of toolkit components are still under development)

Outline

New Usability Tools Make CCA More Attractive

- Bocca
- OnRamp
- **Babel's Contracts**
- **Dimensional Units**

Bocca Automates Component Creation

- Bocca can generate operational component code because component interoperability, connectivity, and modularity is independent of function
- This means you only need to provide the implementation
  - Given an implementation language, a name, and port types, a "shell" component can be generated
  - Can import implementations from other components, other code
- Similar in spirit to *Ruby on Rails* for web applications

Bocca Team

Boyana Norris (ANL)
Ben Allan (SNL)
Wael Elwasif (ORNL)
Rob Armstrong (SNL)

OnRamp: Semi-Automatic Wrapping of Legacy Code into Components

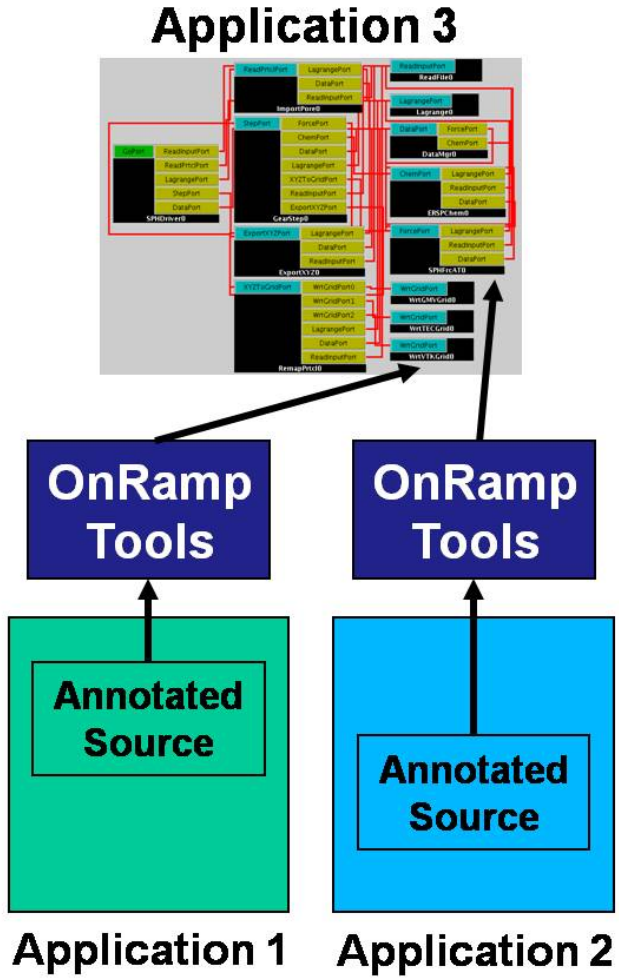
- OnRamp converts existing traditional code into CCA components, using a combination of:
  - Source code analysis
  - Source code annotations
  - Code generation tools
- Overcomes a major practical barrier to CCA adoption
  - Manual wrapping is tedious, and “one-way”

OnRamp Team

Geoff Hulette (UO)
Ben Allan (SNL)
Matt Sotille (UO)
Rob Armstrong (SNL)
Boyana Norris (ANL)

OnRamp Use Cases

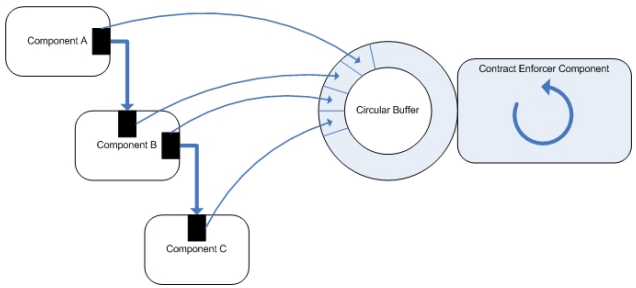
- Simplifies generation/maintenance of CCA bindings for large (legacy) code bases
- Allow actively developed codes to be used in traditional and component environments simultaneously
- Provides a CCA “meeting place” for developers of disparate codes wishing to collaborate
- Currently at “alpha” stage



Interface Contracts

- A set of rules that checked at runtime to ensure correctness
  - Preconditions, postconditions and invariants
  - A way to capture runtime application errors
    - Especially useful when components are developed by separate groups
  - Fully implemented in Babel
- A way to add more semantic information to an interface expressed in SIDL
- Research challenge: adapting contracts to HPC requires that their performance impact be reduced

Offloading Contract Enforcement



Goal: Develop a system to offload executable interface contracts in order to reduce their overhead and increase their acceptance in the high performance computing community

- Offloading can greatly reduce contract overhead
- But be careful, as contracts have to be copied to be offloaded
- Best way is to be selective, only computationally demanding contracts should be offloaded

Team

Kostadin Damevski (VSU)
Hui Chen (VSU)
Tamara Dahlgren (LLNL)

Dimensional Units at the Component Level

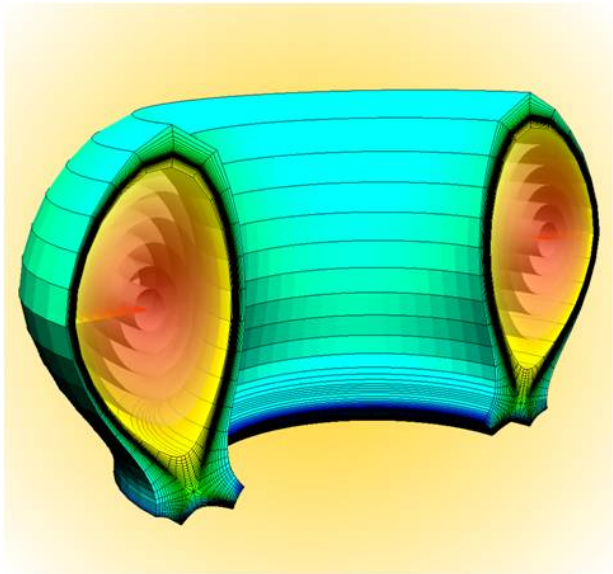
Ongoing research at a relatively early stage...

- Expressing dimensional units (e.g. meters, seconds, newtons, etc.) is not commonplace in scientific programming
  - Big part of paper and pencil calculations
  - Dimensional mismatch errors do exist in practice (e.g. Mars Orbiter)
- Libraries targeted to certain programming languages do exist, but none have gained wide acceptance
- Observation: dimensional mismatch does not occur on the small scale, but as a result of mistaken assumptions in combining code from separate developers
- Plan: add unit information to interfaces

Outline

FACETS Uses CCA Tools for First Integrated Core-Edge Plasma Simulation

- Framework Application for Core-Edge Transport Simulations, PI John Cary (Tech-X), www.facetsproject.org
- Integrated modeling of plasma core, edge, and wall: prototype for Fusion Simulation Project
- Complex physics with different dimensionalities

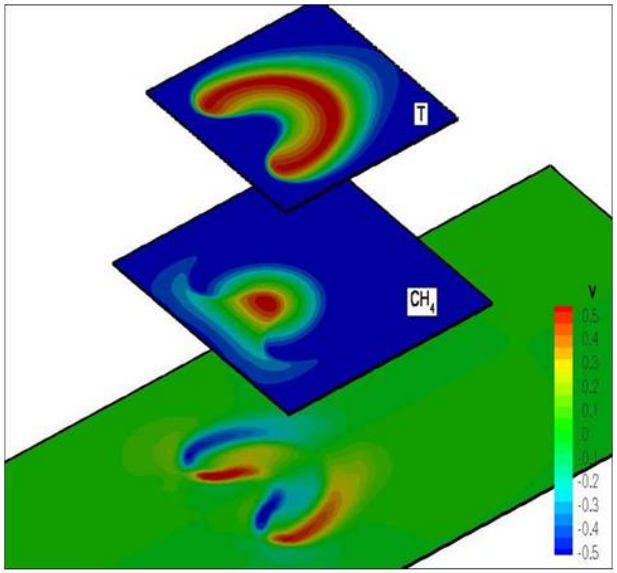


Courtesy of John Cary (Tech-X)

- Use SIDL to express interfaces between components (core, edge, wall)
- Use Babel to integrate UEDGE code with core model in custom FACETS framework

CFRFS Demonstrates Toolkit in 4th Order in Space AMR Simulations

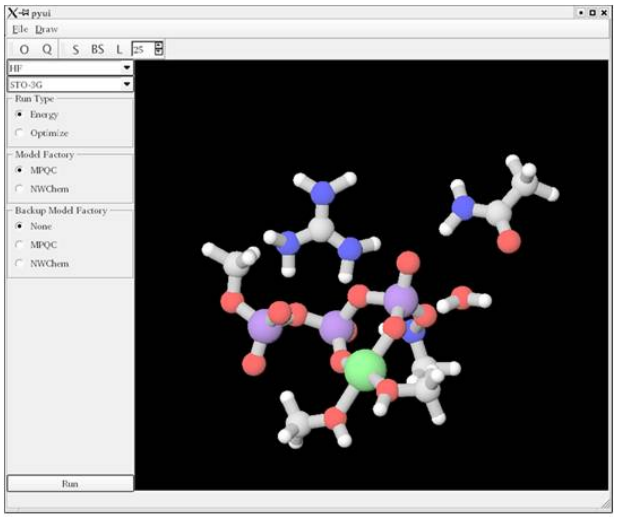
- Computational Facility for Reacting Flow Science, PI Habib Najm (SNL), www.ca.sandia.gov/cfrfs
- CFRFS has developed an extensive CCA-based toolkit for simulation of reacting flows with detailed chemistry
- Integrates contributions from many researchers
- Allows a wide range of simulations to be created in plug-and-play fashion



Courtesy of Cosmin Safta (SNL)

QCSAP uses Components to Enable Interchangeability and Interoperability in Quantum Chemistry

- Quantum Chemistry Scientific Application Partnership, PI Mark Gordon (Ames Lab)
- Representing major chemistry software packages: GAMESS, MPQC, NWChem
- Using CCA to expose unique capabilities of packages to allow use by other packages
- Extending scientific capabilities by creating interoperable community software



Courtesy of the QCSAP project

Summary

- Component technology has been used widely to support:
  - Decompose application into modular “pluggable” parts
  - Provide an adequate basis for reuse
  - Crearly separate interface from implementation
- CCA applies this technology to high performance computing
  - Support for scientific apps: parallelism, scientific types, multidimensional arrays
  - Low overhead
  - Ample tool support
  - Consideration for legacy code bases
- More CCA info can be found at:
  - *http://www.cca-forum.org*
  - *http://tascs-scidac.org*
  - *http://www.cca-forum.org/dev*

Acknowledgements

- CCA Forum
  - David Bernholdt, PI
- DoE SCIDAC program
- Xiaolin Li and Jim Jiao for inviting me

QUESTIONS?